SDC – 93 – 573
UM – HE – 93 – 29

# UNIDAQ

Version 2.3

Software for UNIX-Based Data Acquisition

# User's Guide

Editor: R. Ball

September 26, 1995

*University of Michigan*
*National Laboratory for High Energy Physics (KEK)*
*Tokyo Institute of Technology*
*University of Minnesota*

Authors may be reached as follows:

bob.ball@umich.edu
timmer@mnhep1.hep.umn.edu
nomachi@kekvax.kek.jp

# 1. INTRODUCTION

This document is part of an overview of the UNIDAQ Data Acquisition System, Version 2.3 . The full suite of documents includes an Installation Guide, a Technical Manual, and a User's Guide (this document). All three are available in postscript format in the doc sub-directory of the distributed software. The user is also referred to Fermilab PN457, the Murmur User's Guide (pn457.ps.Z in the directory containing the distribution tar files.)

The overall system software consists of a generic *template* structure, a *buffer manager*, and a set of *processes* such as a collector, analyzer, recorder, and runcontrol, which were developed using the template and the buffer manager. The processes running in UNIDAQ interact with each other by means of three entities: *command messages*, *event data*, and *process data.* The interprocess message passing is handled by the template, the circulation of event data is controlled by the buffer manager (NOVA), and the process data are handled by the Status Path. The interaction of the processes is illustrated in Figure 1.

Two of the criteria considered in the design of the UNIDAQ Data Acquisition System are *extendibility* and *scalability.* The template is extendible by providing a generic underlying structure on which the processes are built. The processes built on top of the template inherit all the characteristics provided by this generic structure, and if required, additional process-specific features can be added to a particular process without having to modify the kernel template. This facilitates the addition of new processes to the system. Similarly, both the template and the buffer manager are scalable up to distributed environments and higher event rates, and allow on-the-fly addition and deletion of processes.

In this report we describe how the average user can take the installed software, start up the system (assuming it has already been configured for the hardware in use) and analyze and record data. Later sections detail how the software can be configured for the particular set of modules the user would like to read out.

## 1.1 What is UNIDAQ?

UNIDAQ is a reasonably portable, modular, UNIX based, data acquisition system. Its implementation for a given user can range from a very simple, read data / write data two process system, all the way up to a system which runs on multiple processors with multiple data sources and a variety of analysis and display activities.

UNIDAQ is also ported to the real-time VXWORKS and LynxOS operating systems, though not all processes are available on these systems and there are some small differences in starting processes as will be pointed out later.

Data in the form of events is passed using the buffer manager system NOVA developed at KEK and TIT.

Commands in the form of messages are passed using the template model developed at the SSCL.

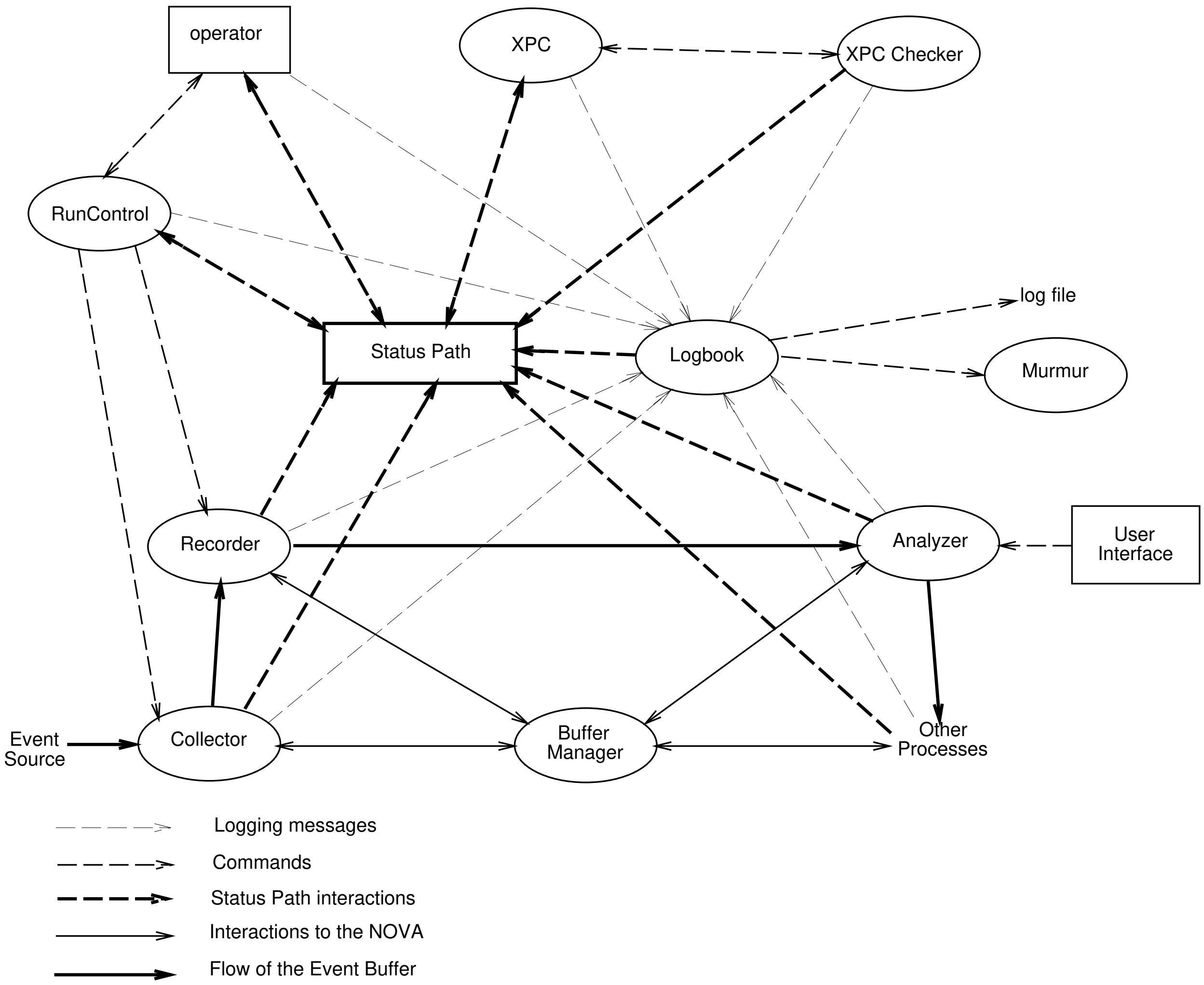


Figure 1. Interaction of the processes.

The overall system status can be monitored and controlled using the Status Path and control processes developed at the University of Michigan.

A corollary of the Status Path is critical variables. Critical variables are those system and user level variables which are thought to be important enough to retain between invocations of a process, whether that process stops normally or dies abnormally.

The user can choose those processes she/he needs to perform a task and run only those. Others can be added or dropped on the fly as they are needed or no longer required.

The following chapters will give a brief introduction (and sometimes not so brief) to all these pieces. Further details can be found in the accompanying Technical Guide for those who want or need such information.

### 1.2 Suggested Reading

Suggested minimal reading in this User's Guide includes Chapters 1, 2 and 7, and those sections of Chapter 3 dealing with the processes you intend to run. If you intend using multiple processors or VXWORKS you should also read Chapter 6. Chapter 5 contains a detailed sample session using UNIDAQ which can be very instructive.

### 1.3 Acknowledgements

The editor would like to thank all those who contributed text, figures and ideas to this document. These persons include, but are not limited to, R.Ball, C.Timmermans, A.Fry, D.Brenner, C.Erbas, M.Nomachi and B.Roe. E-mail help can be obtained on this package from:

| | |
|---|---|
| MICH::BALL | bob.ball@umich.edu |
| MICH::TIMMER | timmer@mnhep1.hep.umn.edu |
| KEKVAX::NOMACHI | nomachi@kekvax.kek.jp |

## 2. RUNNING UNIDAQ V2.3

With the software installed on your machine as instructed in the Installation Guide you can now begin to play. This chapter explains how to set up the environment needed to run the software, how to start it all, how to stop it all, and how to actually take data using the various processes.

### 2.1 Setting Up

The UNIDAQ software makes use of a number of environmental variables (c-shell) during its operation on UNIX platforms. A file in the UNIDAQ directory, $USERUNIDAQ/conf/setup.csh, contains the majority of these variables. A few others are needed if murmur is running, but use of these is transparent to the casual user. To set these variables it is recommended that your .cshrc file in your home directory be modified to access this file whenever you log in. The .cshrc file was chosen over the .login file because the .login is not really used in the HP VUE environment. Add to the .cshrc file the line (this file is actually a link to the file in the conf directory)

```
source $USERUNIDAQ/setup.csh
```

Of course, the variable USERUNIDAQ may not be set up yet, so you should ask the person who installed the software which directory it corresponds to, and substitute that directory for $USERUNIDAQ in the line you add. Typically this directory will be something like /tr2/UNIDAQ/v2.3/user_UNIDAQ. See the Installation Guide if you wish to make your own, mostly links copy of the UNIDAQ directories. In this case $USERUNIDAQ will point to the directory you choose. It is also possible to just create your own local copies of the user directories and work within them using the "user_tree" command in the $TOP-UNIDAQ/bin directory. Again, refer to the Installation Guide for these directions.

As a result of sourcing `setup.csh`, a group of environmental variables will be set, your PATH variable will be modified to search the $USERUNIDAQ/bin and $TOPUNIDAQ/bin directories for the UNIDAQ user and system processes, and the MANPATH variable will be modified to point to some man page entries for the UNIDAQ system.

With distributed UNIDAQ, the possibility exists to run processes on several machines. To enable this, you must have a ".netrc" file in your home directory, listing the machines and accounts you will use. See the Technical Guide for details on the format of this file. The machines themselves are listed in a separate file, $USERUNIDAQ/conf/nodefile, and a startup file (described later and in the Technical Guide) should be prepared for each of these machines.

### 2.1 a. Configuring

Some UNIDAQ processes and files can and should be configured according to the needs of the experiment. The uni_config tool provides an easy way of doing so. A description of this tool can be found in the chapter "ADDITIONAL TOOLS". Furthermore, you are referred to the chapters in this manual and in the Technical Guide on the processes you want to modify.

## 2.2 Starting the Software

The simplest way available to start all the software on UNIX machines, once you have sourced the setup.csh file, is to simply type the command

Start

at the shell prompt. This executable image, which resides in the $TOPUNIDAQ/bin directory, will start the user and control processes on all processors associated with UNIDAQ and listed in the `nodefile` file (located in $USERUNIDAQ/conf). This is accomplished by finding shell scripts in this same directory called Start_`node`, where `node` corresponds exactly to the machine name as it is listed in the `nodefile` file. The processes named in the script will be started on the corresponding node. The *uni_config* tool can be used to edit all these files.

On a combined UNIX-VXWORKS system, the Start-command will not work correctly (although it will not crash). Processes will only start on the UNIX machines, while nothing happens on VXWORKS. The Start-scripts have to be executed manually on VXWORKS (see below), or can be executed at boot-time of the VXWORKS systems.

The processes on a single UNIX node can be started by logging into that node and typing the command

Start_local

In a single processor configuration, typing "Start_local" works as well as typing "Start" to get the processes going. See the Technical Guide for more details on this procedure.

On a VXWORKS node one has to type <`Start_node`, where "node" describes the full node.network address of the VXWORKS machine. *Start_local* does not work here.

The user processes consist of *collector, analyzer, recorder, dataview* and *UNIview*, respectively the processes which read out data, manipulate/make histograms from the data, write the data to media, display selected event variables on your workstation, and display a snapshot of the UNIDAQ machine/process tree. Actually, *analyzer* and *UNIview* are not started because they are interactive in nature. Furthermore *analyzer* is actually a suite of several different programs, similar in nature and functionality; these will be explained in later sections. One further user process called *receiver* exists to collect NOVA events across the network and place them in a local NOVA buffer (see Chapter 6, section 1 for more on network NOVA). Of these user processes only *collector, recorder* and *receiver* are available on VXWORKS.

The control (or system) processes consist of *XPC, xpc_checker, logbook,* and *runcontrol.* One further control process, the *operator* process, is not automatically started. This process runs interactively with the user to control the operations of the system as a whole. None of these control processes are available on VXWORKS.

If TCL/TK is used, *operator*, *dataview* and *uni_config* are implemented as tcl-scripts. *Operator* and *dataview* are based upon *UNIwish*, a "wish"-shell modified for UNIDAQ, as described below. One additional process, *UNIview*, exists only as a tcl-script.

Finally, to enable messaging across the network a control process called *msg_server* is started. At a minimum one XPC (on UNIX machines) and one *msg_server* (on both UNIX and VXWORKS machines) should be started on each node listed in the $USERUNIDAQ/-conf/nodefile file.

### 2.2 a. Command line options in starting processes

Three command line options are available when starting processes which use the template.

1. `-m machine` — use this option to specify the process should get its NOVA buffers from machine `machine` instead of from the local processor.
2. `-n name` — use this option to specify the name of the process as registered in the Common Table and as seen by other template-using processes. Normally the process name is just the name of the program image.
3. `-p prio` — specify the priority at which this process will receive NOVA buffers. The allowable range is 1-100, and will be adjusted appropriately if the value supplied is out of this range. Higher priorities receive buffers earlier in the hand-down chain. Values above 70 ensure buffers will all be received, whereas lower values may mean missed buffers in a busy system.

There are differences in starting processes on a UNIX machine and on VXWORKS. On UNIX one can type

process_name options & ,

or alternatively

nohup process_name options & .

On VXWORKS one first has to load the file using

ld <process_name ,

before starting the process by

sp process_name, "opt1", "opt2", ...

For example starting a collector process with a different name would look like:

sp collector, "-n", "new_name".

## 2.3 Stopping the Software

The simplest way available to stop all the software on UNIX machines once you have sourced the setup.csh file is to simply type the command

Reset

at the shell prompt. This executable image, which resides in the $TOPUNIDAQ/bin directory, will stop all processes on all UNIX processors associated with the UNIDAQ system, and clean up the shared memory and message queue entries which were allocated. To stop only those processes on a single UNIX processor, log into that processor and type the command

Reset_local

"Reset_local" completely clears the local processor of processes, shared memory and message queue entries without adversely impacting other processors in the system. Both "Reset" and "Reset_local" prompt the user for confirmation. In a single processor configuration, "Reset_local" works as well as "Reset".

On VXWORKS machines these "Reset" tools are unavailable; each process has to be stopped manually using the *ask* tool. Another way of stopping processes is using the UNIX/VXWORKS kill command.

## 2.4 Interacting with the Software

Once you have started all the processes as above, the system will be sitting in an idle state waiting to be told what to do. To get it moving there are two possibilities. Either one can run the *operator* process, or the *ask (rask)* tool can be used to send commands directly to the various processes. Note that from a VXWORKS platform, only the *ask* tool can be used.

The *operator* program uses X-based displays and must be run such that it can display at a workstation. The look and feel of this process depends on whether TCL/TK is used or not. The information given to the user and the possibilities of sending commands are the same in both cases. To invoke it enter at the shell prompt the command

`operator`.

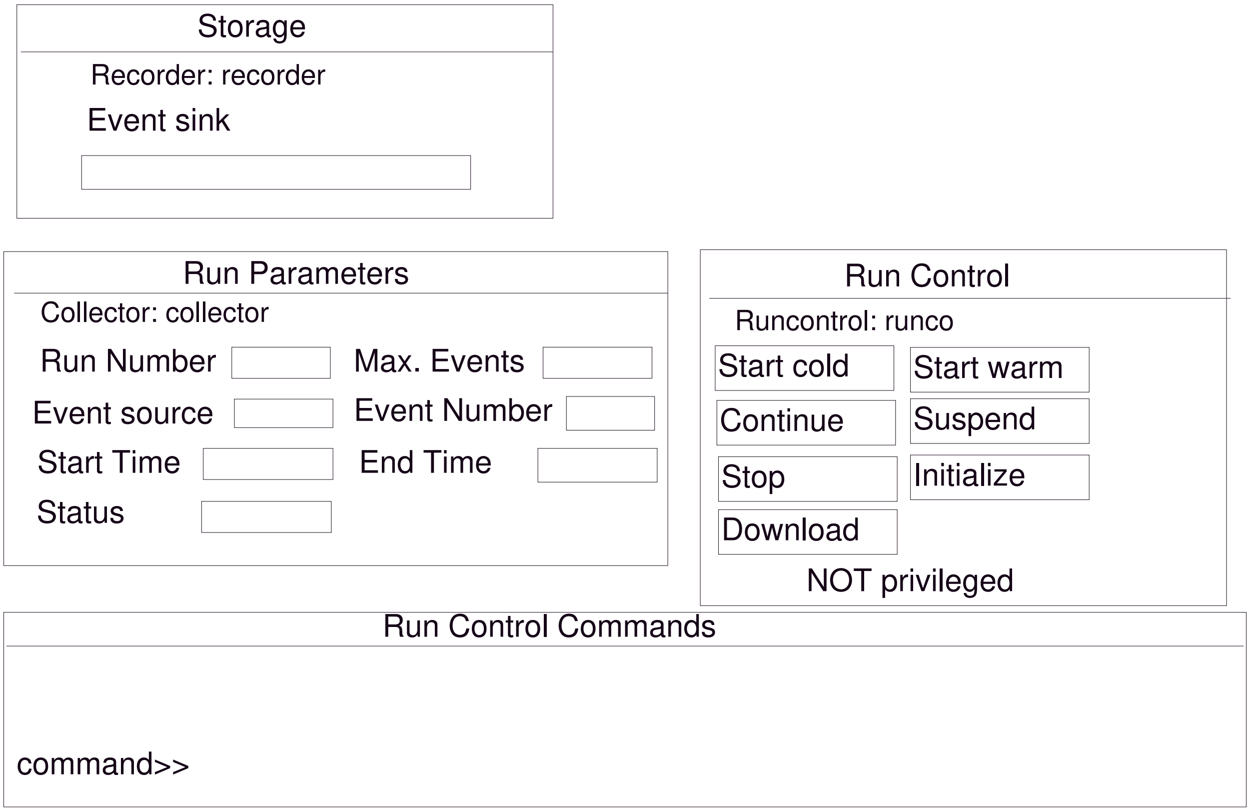


Figure 2. The Four Default *Operator* Process Windows.

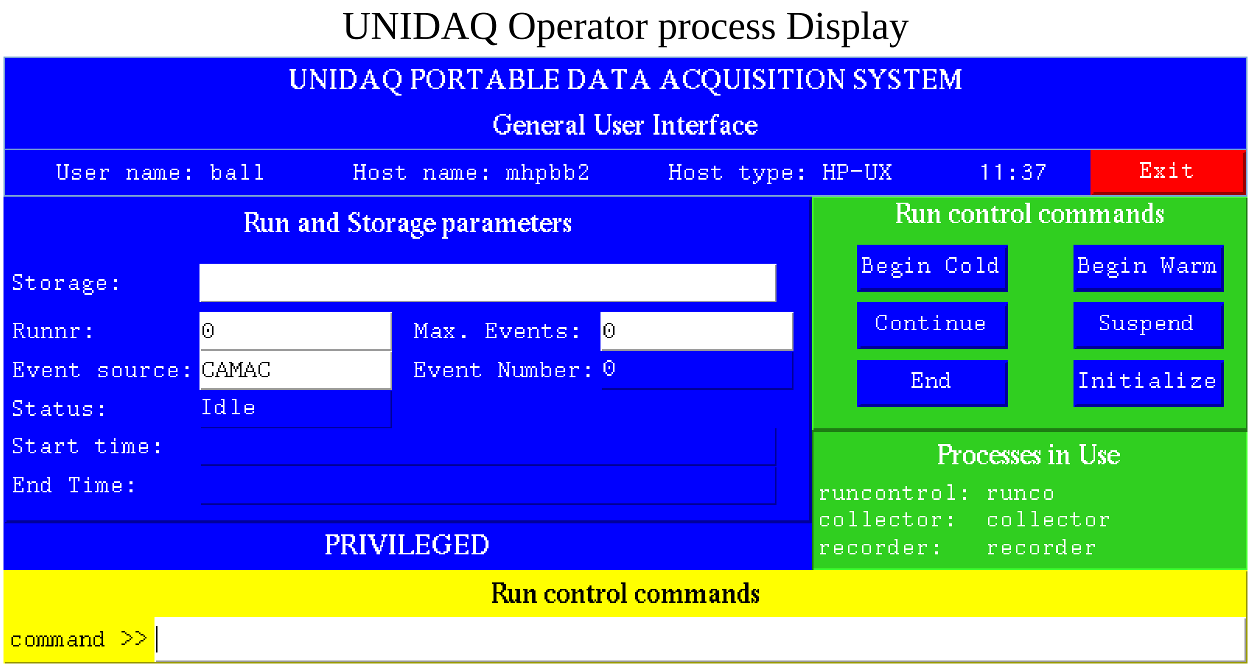


Figure 3. The *Operator* Process Window Under TCL/TK.

You could run this in the background as well via the command "operator &".

In the non-TCL/TK version of *operator* four windows will appear on your monitor. Drag these apart if they overlap each other so that you can see what is going on with them. These four windows are shown in figure 2. One window is a command-line input window, one is used to specify the directory in which *recorder* should record its outputfile, one is used to specify the data source (currently CAMAC or PSEUDO) and doubles as a statistical display, and one is the primary control panel with buttons for various operations. Placing the cursor within a button and clicking the left mouse button is the method of selection. Runs are begun and ended using this "point and click" method.

The TCL/TK version of operator will only open 1 window on your screen as shown in figure 3. This single window contains the same information as described above for the non-TCL/TK version of operator. Furthermore, it has the same functionality.

Please see the *operator* process description below for further details on using it.

As the software is distributed the *collector* process will not correspond to the CAMAC/VME hardware which you wish to read out. You will have to modify the source code of *collector* to properly correspond for you. Chapter 5 in this manual and a chapter in the Technical Guide give examples of how to modify the program code and rebuild it to your needs. After this rebuild, the software can be completely restarted as above, or just collector restarted, and data taking can commence.

### 2.5 Stopping Single Processes

Excepting only the buffer manager (NOVA) all the processes recognize the EXIT command as this is built into the message template. Because of this processes usually do not have to be killed via Unix/VXWORKS "kill" command, which is messy, but can be cleanly stopped using the *ask (rask)* tool. Given the name of the process (available by typing `status` at the shell prompt) one can simply issue the command

ask *process_name* EXIT

to the shell. The process will stop, cleaning up all the UNIDAQ common tables in the process. On UNIX machines, it can be re-started later using the *ask* command, requesting the XPC to take care of the task. See the discussion of the XPC process below for details of this. On VXWORKS the standard "sp" command should be used.

### 2.6 Looking at Event Data

When the data flow is started (*e.g.*, using *operator*) several possibilities exist to look at the event data.

The *analyzer* processes can be used to create online histograms. Up to three *analyzer* versions are available, depending on which platform is in use. *Ana_d* and *ana_nd* are linked to include PAW. The *ana_gl* program creates histograms in shared memory and one has to run PAW separately to display the histograms.

The *dataview* process allows one to show current values of variables on the screen. These values are automatically updated as the dataflow progresses.

An alternative way of looking at event data is provided by the *UNIdump* tool. This tool, which is described in chapter 7, provides an easy way to display the data on an event by event basis.

# 3. SYSTEM PROCESS DESCRIPTIONS

In Chapter 6 of the Technical Guide we describe the template. The template provides the generic structure on which the processes can be built. UNIDAQ 2.3 contains ten such processes. Five of these processes (*collector*, *recorder*, *receiver*, *analyzer*, and *dataview*) are aimed to deal with events. Five processes (*runcontrol*, *logbook*, *operator*, XPC and *xpc_checker*) are targeted to control the operation of the system. As mentioned before, some processes can be programmed (*operator*, *dataview*, and *uni_config*) as tcl-scripts. This required the use of a modified TCL/TK "wish"-shell called *UNIwish*. An additional process called *UNIview* exists only as a tcl-script. One further process, the *msg_server*, enables communications between machines. In this chapter, we describe these processes, and the *UNIwish* shell.

Note that the User-level Commands and Variables are "values" which the process can understand in messages that it receives. For example, the *collector* process, described next, understands the command RESUME. Using the command line message sender program **ask** then one can send the following command to *collector*:

ask collector RESUME [optional message string]

The *collector* process, when it receives this message, will take the action specified below. A program level subroutine, Send_Message, also exists to send the same commands.

## 3.1 Collector Process

*Collector* is the process which interacts with the event source. It waits until an event occurs, and then accesses the event source and reads the channels. The event data is then stored into a buffer, and the buffer address is given to the next process in the process chain.

If *collector* gets stuck for a long period of time while processing an event, or during setup, two ways are provided to break it out of this state. The first is to code the program such that it checks the "interrupt_flag" global variable for a non-zero value, and return if such is found (see the description of the INTERRUPT command of the XPC process). The second is to send an interrupt to the process via the UNIX/VXWORKS command "kill -INT <pid>" which will immediately return the process to the main template loop; this is different than the default behavior described in Chapter 9 of the Technical Guide.

In UNIDAQ 2.3, the *collector* provides the following user-level commands and variables:

Collector User-level Commands:

BEGIN, RESUME, PAUSE, END, FLUSH, MSET, and EXIT.

Collector User-level Variables:

runnr, maxnevents, eventnr, eventsource, runstate, starttime, endtime, runtype, mode, flush_all.

Collector Critical variables

All user-level variables, with the exception of runstate, are critical.

### 3.1 a. User-Level Commands

The descriptions of the *collector* user-level commands are given below:

**BEGIN** *<runnr>* *<comment>*: This command is used to start a run. Here, *<runnr>* is used as the identification number of the run, and is stored into the *collector* user-level variable of that name. *<comment>* is a string which is packed into the Begin_Run record. Whenever the *collector* receives a *BEGIN* command it initializes the event source by sending hardware related signals, and generates a Begin_Run record. However, the data taking process does not start with the *BEGIN* command. If *<runnr>* is not supplied, then the current value of the user-level variable is used. Zero is a legal value for *<runnr>*. If a *<comment>* is supplied, then the *<runnr>* must also be supplied; no check is made to enforce this. The default comment is "No comment.".

**RESUME** *<comment>*: This command is used to resume a run. The actual data taking process starts with this command. After receiving the *RESUME* command, the *collector* enables the interrupts coming from the event source, generates a Resume_Run record, and waits for an event. The default comment is "No comment.".

**PAUSE** *<comment>*: This command is used to pause a run. Whenever the *collector* receives a *PAUSE* command, it disables the interrupts coming from the event source, and generates a Pause_Run record. The default comment is "No comment.".

**END** *<comment>*: This command is used to terminate a run. After receiving an *END* command, the *collector* resets the event_source, and generates an End_Run record. The default comment is "No comment.".

**FLUSH:** This command forces the current NOVA buffer to be flushed whether it is full of events or not, thus passing it to the next UNIDAQ process.

**MSET** *<var1=value1>* ... *<varN=valueN>*: This command allows multiple variables to be set in a single command. Otherwise it acts in all respects the same as the system command SET.

**EXIT:** This command calls the exit handler, cleans up the message queues and semaphores used by the *collector*, updates the common status table and then exits.

### 3.1 b. User-Level Variables

The descriptions of the *collector* user-level variables are given below:

<*runnr*>: Integer. Initial value is zero. This is the run number of the current run. It may be changed between runs, but will take effect only at the beginning of the next run.

<*maxnevents*>: Integer. Initial value is zero. This is the maximum number of events which a run may have before *runcontrol* (if it is being used) will automatically end it.

<*eventnr*>: Integer. Initial value is zero. This is the current event number within a run. It resets to zero at the beginning of a run.

<*eventsource*>: String. Default value is "CAMA". This is the source of data events. At this time, recognized values are "CAMAC" and "PSEUDO" for events out of CAMAC and a random number generator. Other values are allowed, but no specific code exists for them.

<*runstate*>: Integer. Default value is END_STATE. This value reflects the state of data collection in an obvious fashion. Other values are RESUME_STATE, BEGIN_STATE, and PAUSE_STATE. These four respectively take the values 0, 3, 1, and 2. This variable is used to assure an orderly transition between the various data collection states of the collector, so that buffers are written to tape in the correct order.

<*starttime*>: Integer. Default value is zero. The time at which a run begins.

<*endtime*>: Integer. Default value is zero. The time at which a run ends.

<*runtype*>: Integer. Default value is zero. This variable is free for use by the programmer. It is intended to indicate the type of run, eg, pedestal, calibration, etc.

<*mode*>: Integer. Default value is zero. This variable is free for use by the programmer. It is intended to indicate some particular setting within a <*runtype*>, for example, for a calibration run it could indicate charge injection, pulse, etc.

<*flush_all*>: Integer. Default value is zero. If this variable is non-zero, then all event buffers are flushed containing only one event. This is useful when testing, but inefficient during normal data taking.

## 3.2 Analyzer Process

At this time there are three versions of the *analyzer* process. As many versions as possible are implemented on each platform, but all of them are not always available, depending on the limitations of the hardware and software. The version described next, the dynamically linked *analyzer* called *ana_d*, is available on the HP, SGI and SUN platforms. A non-dynamically linked version of this same program (*ana_nd*) is available on the same three platforms in addition to the DECstation. The third version (*ana_gl*) uses a global section to store HBOOK histograms, and is available only on the DECstation and SUN platforms. This version is run in association with the standard PAW program for viewing the histograms. Off-line versions of the first two (*ana_d_off* and *ana_nd_off*), which run independently of the other UNIDAQ processes, are available on the same platforms as their on-line cousins. No *analyzer* process is available on VXWORKS or LynxOS.

3.2 a. The Dynamic Analyzer

*Ana_d* is used for on-line histogramming of the event data. *Ana_d* provides all the features of PAW, so the user can use the *ana_d* as a PAW screen. In fact, you are automatically connected to a "paw>" prompt in this *analyzer* version. Further, it allows the user to define his/her own histograms using PAW and dynamically link those histograms to the system. The user defines his/her own analysis procedures using FORTRAN or C. These procedures can then be compiled "on-the-fly" and dynamically bound to an executing *ana_d* process. After the histograms are linked to the system, the incoming event data is used to fill the histograms.

As with the *collector*, the *ana_d* can be sent back to its main template loop by the UNIX command "kill -INT pid". This is useful in case the user has set up a bad/looping routine.

In UNIDAQ 2.3, the *ana_d* provides the following user-level commands and variables:

Ana_d User-level Commands:

LINK, UNLINK, ADD, DELETE, LIST, DISPLAY, PAW, PAWC, BEGIN, END, COMPROC, HLIMAP, QUIT, and EXIT.

Ana_d User-level Variables:

runstate, run_number.

Ana_d Critical Variables:

None.

User-Level Commands

The descriptions of the *ana_d* user-level commands are given below:

**LINK** <*file_name1*> [ <*proc_name1 ...*>] [ <*file_name2*> [ <*proc_name1 ...*>] ] ...: The *LINK* command allows the user to compile and dynamically bind a file or files <*file_nameN*> to the *ana_d*. If the <*proc_nameN*> arguments are also supplied, then the **ADD** command is also executed for the named procedures. This command is typically used to link a procedure filling a PAW histogram. More than one file can be linked to the *ana_d* during one run. When the *ana_d* receives the *LINK* command, it first compiles the file then links the object file. Finally it opens the linked file so that the new routines are now a part of the running image.

**UNLINK** <*file_name*>: This command is used to unlink a file <*file_name*> which has already been linked to the *ana_d*. Further, if the user wants to unlink all the linked files, he/she can use the *UNLINK ALL* command.

**ADD** <*file_nameN*> <*proc_name 1*> ... <*proc_name n*>: Once a file <*file_name*> has been successfully linked, the user specifies the procedures <*proc_name 1*> ... <*proc_name n*> to be executed in that file. This is done by the *ADD* command. After the *ana_d* receives the *ADD* command, it calls the procedures for every incoming event data. More than one file, each with its own entry points, may be specified on a single **ADD** command.

**DELETE** <*file_name*> <*proc_name 1*> ... <*proc_name n*>: The *DELETE* command allows the user to delete a procedure that has been added using the *ADD* command. The delete command has three basic formats. The first format is used to delete one or several procedures from the same file. To do this the user gives the file <*file_name*> and the procedures <*proc_name 1*> ... <*proc_name n*>. The second format *DELETE* <*file_name*> *ALL* can be used to delete all the procedures belonging to a file. The third format *DELETE ALL* deletes all the procedures that have been added so far to the system.

**LIST:** List all files linked to the *ana_d* image and their ADDed procedure names.

**DISPLAY** <macro_name> <*time_interval*>: This command is used to invoke a kumac macro <*macro_name*> periodically. Generally, the macro is used to display the histograms filled on the screen. The <*time_interval*> gives the length of the time interval in seconds between two displays. If <*time_interval*> is not given, then the default time interval of 10 seconds is used. Whenever a new *DISPLAY* command is issued the previous *DISPLAY* command is cancelled. If the user wants to cancel the previous *DISPLAY* command, and does not want to issue a new macro, he/she can use the *DISPLAY OFF* command.

**PAW** <*paw_command*>: Commands can be issued to the *ana_d* in two different ways: from the user command line of the *ana_d*, or from another process. The *ana_d* user command line is directly connected to PAW, so the user can directly issue <*paw_command*>'s. However, if another process wants to issue PAW commands to the *ana_d*, it should send the command PAW <*paw_command*> via the "Send_Message" subroutine.

**PAWC** <*paw_command*>: The action is the same as for the PAW command except this version is used internally by the *ana_d* and forces parent-child process synchronization through the use of semaphores.

**BEGIN** <*runnr*>: This command may be used to start or override a normal run. Every run should be started by a *BEGIN* command. Here, <*runnr*> is used as the identification number of the run. Whenever the *ana_d* receives a *BEGIN* command or a BeginRun_Record, it executes the PAW command "*exec anbegin.kumac* <*runnr*>". The value of <*runnr*> is stored in the user-level variable *run_number*. If *ana_d* receives a BeginRun_Record from NOVA, the run number in that record is stored in <*run_number*> and a *BEGIN* command is executed.

**END:** This command may be used to terminate a run. Whenever the *ana_d* receives an *END* command or an EndRun_Record, it executes the PAW command "*exec anend.kumac*". The run number is reset after each *END* command.

**COMPROC** <*filename*> or @<*filename*>: This command is used to execute a file <*filename*> of *ana_d* or other template-related commands. <*filename*> may be in the directory from which *ana_d* was started, or it could be an absolute pathname, or it could begin with an environmental variable.

**HLIMAP** <*global_name*>: Make a global memory of histograms of name <*global_name*> so that other PAW images may access them. This is the same way in which the *ana_gl* works. THIS COMMAND IS NOT WORKING AT THIS TIME.

**EXIT** or **QUIT:** Whenever the *ana_d* receives the EXIT command, it calls the exit handler, cleans up the message queues and semaphores used by the *ana_d*, updates the common status table and exits.

Next to these commands, *ana_d* has some history commands available, which resemble the c-shell history commands.

**!**: Displays the list of issued commands, to the maximum history.

**!!**: Recalls the last issued command.

**!N**: Recalls command number N from the history.

**!string**: Recalls the most current command beginning with "string".

User-Level Variables

The descriptions of the *ana_d* user-level variables are given below:

<*runstate*>: Integer. Default value is END_STATE. This variable reflects the type of the last event buffer analyzed by the process. Other values are RESUME_STATE, BEGIN_STATE, and PAUSE_STATE. These four respectively take the values 0, 3, 1, and 2.

<*run_number*>: Integer. Default value is -1. This variable reflects the current run number within UNIDAQ. The user has no particularly good use for it.

### 3.2 b. The Global Section Analyzer

The global section version of *analyzer* called *ana_gl* creates histograms in shared memory. Different PAW sessions can access this shared memory at the same time, which allows multiple users to have access to the same data. Users need to create their own tailored version of *ana_gl* as described in the UNIDAQ technical guide. The name of the shared memory opened by this *ana_gl* can be retrieved from the programmer, or if you want to browse through the code, this name is an argument in the hlimap() call (check the HBOOK manual). The distributed version of this program uses the name TMC. When you know the name you can start PAW. Within PAW you type:

```
paw> global shm_name
```

where shm_name is the name of the shared memory opened by the *ana_gl*. Although you cannot list the histograms and/or ntuples created in this shared memory you have access to them. Again, the *ana_gl* programmer needs to tell you which histograms and ntuples are created.

In UNIDAQ v2.3 *ana_gl* provides the following user-level commands and variables:

Ana_gl User-level Commands:

BEGIN, END, and EXIT.

Ana_gl User-level Variables:

runstate.

Ana_gl Critical Variables:

None.

User-Level Commands

The descriptions of the *ana_gl* user-level commands are given below:

**BEGIN** <runnr> : Whenever *ana_gl* receives this command it sets the run_number internal and the runstate user-level variables. If *ana_gl* receives a BeginRun_Record from NOVA, the run number in that record is stored in <*run_number*> and a *BEGIN* command is executed.

**END:** This command ends the current run. *Ana_gl* sets the runstate.

**EXIT:** *Ana_gl* calls the exit handler, and terminates the connection with NOVA.

User-Level Variables

The descriptions of the *ana_gl* user-level variables are given below:

<*runstate*>: Integer. Default value is END_STATE. This variable reflects the type of the last event buffer analyzed by the process. Other values are RESUME_STATE, BEGIN_STATE, and PAUSE_STATE. These four respectively take the values 0, 3, 1, and 2.

3.2 c. The Non-Dynamic Analyzer The non-dynamic analyzer *ana_nd* works in a fashion similar to the *ana_d* process except that analysis routines are linked in the more traditional fashion rather than being dynamically linked into a standard image. Consequently all User-level commands dealing with the dynamic link are unavailable. To assist in preparing this program the *ndmake* tool was created. The user simply creates a file called "NDMakefile" with one command line per analysis file to be linked into the *ana_nd* image, then runs the *ndmake* command. Each line in the NDMakefile contains the name of the Fortran or C file to be linked, and the name(s) of the routines to be called as each event is processed. For example, a line may look like (this is from the distributed NDMakefile)

nd_dummy.f nd_dummy

which tells *ndmake* to compile and link into *ana_nd* the file nd_dummy.f, calling subroutine nd_dummy for each event processed. For a complete help listing on *ndmake*, enter the command "ndmake -h".

In UNIDAQ 2.3, the *ana_nd* provides the following user-level commands and variables:

Ana_nd User-level Commands:

DISPLAY, PAW, PAWC, BEGIN, END, COMPROC, HLIMAP, QUIT, and EXIT.

Ana_nd User-level Variables:

runstate, run_number.

Ana_nd Critical Variables:

None.

User-Level Commands

The descriptions of the *ana_nd* user-level commands are the same as the corresponding commands for *ana_d* above.

User-Level Variables

The descriptions of the *ana_nd* user-level variables are the same as the corresponding variables for *ana_d* above.

3.2 d. Off-line analyzer differences

The two off-line analyzer programs *ana_d_off* and *ana_nd_off* implement three additional User-level commands which are not available in the on-line versions.

**ANALYZE** <*file_name*>: Open (or continue with) the specified *file_name*, and analyze the events it contains.

**PROCESS** <*file_name*>: Synonymous with ANALYZE.

**UCLOSE:** Close the run file currently being analyzed.

### 3.3 Recorder Process

The *recorder* is responsible for writing data in a file. It is both event and command driven, in the sense that the activity of the *recorder* can be controlled either by event data or by command messages. If data is to be recorded to disk (*eventsink* set to "disk") the *recorder* opens a new file based on the run number either passed by the BEGIN command or by a BeginRun record. The file name is obtained by concatinating an 'r' with the zero padded run number followed by the extension '.dat'. If a file with the same name already exists, then a number will be added to the end of the extension. For example, if the run number is 7945, the *recorder* names the file as r0007945.dat. If this file already exists, then the new file is called r0007945.dat2, and so on.

The user can set the path in which the file will be created by setting the *dest_dir* variable. If *dest_dir* is empty, the file is written in the directory from which *recorder* was started.

If *eventsink* is set to "tape", then *device* should be set to the name of the tape device to be opened and used. Event buffers will be packed end to end and written to tape in 60000 byte blocks.

If the user does not want to write data in a file or to a tape, the *eventsink* variable should be set to "dummy".

In UNIDAQ 2.3, the *recorder* provides the following user-level commands, and variables:

Recorder User-level Commands:

BEGIN, END, LOAD, UNLOAD, REWIND, NEW_DRIVE, END_TAPE, and EXIT.

Recorder User-level Variables:

eventsink, runstate, dest_dir, and device.

Recorder Critical Variables:

eventsink, dest_dir, and device.

3.3 a. User-Level Commands

The descriptions of the *recorder* user-level commands are given below:

**BEGIN** <*runnr*>: This command may be used to start or override a run. Here, <*runnr*> is used as the identification number of the run, and in the name of the output file. If no <*runnr*> is given, then by default the *recorder* uses zero as the <*run_number*>. Whenever the *recorder* receives a *BEGIN* command or a BeginRun_Record, it opens the file for writing. If *recorder* receives a BeginRun_Record from NOVA, the run number in that record is stored in <*runnr*> and a *BEGIN* command is executed.

**END:** This command may be used to terminate a run. Whenever the *recorder* receives an *END* command or an EndRun_Record, it closes the file.

**LOAD:** Load the scsi tape indicated by variable *device.*

**UNLOAD:** Unload the scsi tape indicated by variable *device.*

**REWIND:** Rewind the scsi tape indicated by variable *device.*

**NEW_DRIVE:** Same as the LOAD command.

**END_TAPE:** Write a file mark on the scsi tape indicated by variable *device.*

**EXIT:** Whenever the *recorder* receives the EXIT command, it calls the exit handler, cleans up the message queue used by the *recorder*, updates the common status table, closes the file and exits.

3.3 b. User-Level Variables

The descriptions of the *recorder* user-level variables are given below:

<*eventsink*>: String. Initial value is an empty string. If the value of this string is "dummy" (case independent) then no output event file is written. If the value is "disk" then the output event file is opened relative to the value of <*dest_dir*>. If the value is "tape" event data will be written to SCSI device <*device*>. Any other setting results in data being written to disk.

<*runstate*>: Integer. Default value is END_STATE. This variable reflects the type of the last event buffer seen by the process. Other values are RESUME_STATE, BEGIN_STATE, and PAUSE_STATE. These four respectively take the values 0, 3, 1, and 2.

<*dest_dir*>: String. Initial value is an empty string. If <*eventsink*> is not "dummy" or "tape" then this string is used as the initial file/directory specification for the output event file. For an empty string, the file is written in the directory from which *recorder* was started.

<*device*>: String. Initial value is /dev/rst. If <*eventsink*> is "tape" then the event data will be written to this SCSI device.

### 3.4 Dataview Process

The *dataview* process can be used, on UNIX workstations, to display current event parameters on the screen. When TCL/TK is installed, running *dataview* actually means running *UNIwish* with the `dataview.tcl` script. The functionality is similar to the no-TCL/TK process described here.

When the *dataview* process starts, it opens an X-based window on the screen. This window consists of a number of subwindows (to be configured before compiling the process) which contain the current values of the event data.

How to configure this process is explained in the Technical Guide. This process is completely event driven, and has no special user commands or variables.

Dataview User-level Commands:

None.

Dataview User-level Variables:

None.

Dataview Critical Variables:

None.

### 3.5 Receiver Process

The *receiver* process can be used to get data-buffers across the network into a new buffer manager. It requires that *novand* is running in addition to *novad* on the remote machine and *novad* on the local machine. *Receiver* acts as a collector process on its local machine, placing events into NOVA buffers for distribution.

Receiver User-level Commands:

EXIT.

Receiver User-level Variables:

None.

Receiver Critical Variables:

None.

#### 3.5 a. User-Level Commands

The descriptions of the *receiver* user-level commands are given below:

**EXIT:** Whenever the *receiver* receives the EXIT command, it calls the exit handler, cleans up the message queue used by itself, updates the Common Table, and exits.

## 3.6 Run Control Process

The *runcontrol* process, only available on UNIX machines, can be viewed as a command interpreter. It is the means of doing the (possibly) complicated list of actions associated with such innocuous commands as "Begin_Run". The commands and their actions are defined in an input file. The name of this input file is listed in the file 'files.h' located in the $(TOPUNIDAQ)/src/control/runcontrol directory and that name is bound into the *runcontrol* image. This default input-list is named 'command.list' and is located in the $(USERUNIDAQ)/conf directory.

If a command is given (which is not a system, user level, or shell command [see below] ) a child process is created. This child process reads the input file and tries to execute the command. This command can consist of a set of operations that needs to be executed either in parallel or sequentially. If parallel processes are required the child tries to fork more child-processes. If it fails to do so the operations are performed sequentially.

*Runcontrol* is also able to execute shell commands. The command is to be preceded by an @, eg, the message

```
@collector &
```

will start the *collector* process. *Runcontrol* does not fork a child process to do this, so time consuming commands should be started in the background using this message form with the &, as shown in the example.

*Runcontrol*'s functionality depends on the contents of the input file. The file consists of a set of 'macro' definitions. These macros describe *runcontrol*'s reactions to given commands. The 'macro' definition is an ordered list of subroutine calls. Modifications in this file can be made without the need of modifying the *runcontrol* executable. If a new subroutine needs to be written, then *runcontrol* needs to be recompiled and relinked. The macro set distributed with UNIDAQ is sufficient to handle the standard processes and a simple CAMAC hardware situation. See the sub-section below on "Using the operator process" for a list of these macros. Also see the Technical Guide, Chapter 13, for more details.

Runcontrol User-level Commands:

None.

Runcontrol User-level Variables:

runnr, runstate, wait_for, re_prompt, collector, recorder, and operator.

Runcontrol Critical Variables:

collector, recorder, and operator.

### 3.6 a. User Level Variables

<*runnr*>: Integer. Initial value is zero. Before starting a run, the runnr is read from the *collector* process, after which the next runnumber is determined.

<*runstate*>: Integer. Initial value is zero. Its value is read from the *collector* process.

<*wait_for*>: Integer. Initial value is zero. This parameter is used for communication between parent and child processes. It is used to flag if information from other processes has arrived.

<*re_prompt*>: String. Initial value is an empty string. This parameter is used to store the result of prompting a process for information.

<*collector*>: String. Initial value is "collector". This variable determines which *collector* process is in use.

<*recorder*>: String. Initial value is "recorder". This variable determines which *recorder* process is in use.

<*operator*>: String. Initial value is "operator". This variable determines which *operator* process is in use.

## 3.7 Operator Process

The *operator* process can be used, on UNIX workstations, to interact with UNIDAQ. When TCL/TK is installed, running *operator* actually means running *UNIwish* (see section 3.11 below) with the `operator.tcl` script. The functionality is similar to the no-TCL/TK process described below although everything is located in one window instead of four, and the interaction with runcontrol is a little less in the TCL/TK version, since more checking is done before actually sending commands to runcontrol.

Commands to the *operator* process (such as QUERY_RUNPAR) are configured as part of the *runcontrol* process macros. In essence the person on shift clicks a button, eg, "Begin Run", which action is sent to *runcontrol*. *Runcontrol* finds the commands to the *operator* process (among other operations to be performed) in the macro file and sends them to it, which responds by asking the person on shift for the appropriate parameters.

The (no TCL/TK version of) *operator* process reacts to the following commands;

**QUERY_RUNPAR:** After receiving this command, the *operator* asks new run parameters from the person on shift. The run-number, event source and the maximal number of events in a run can be modified in the appropriate on-screen window boxes. After modification the *operator* sends the new variables to *runcontrol*.

**QUERY_STORAGE:** After receiving this command, the *operator* asks new storage parameters from the person on shift. Again these are entered in the appropriate window boxes. After modification the *operator* sends the new variables to *runcontrol*.

**PROMPT** <*args*>: After receiving this command *operator* displays <*args*> in the command line window (see figures 2 and 3) and waits for a response from the person on shift (ended by a return). Afterwards the response is sent to *runcontrol*, which stores it in the user-level variable `re_prompt` for further use.

Every three seconds the *operator* process reads the new values of the run parameters from the Status Path, and updates the screen. The *operator* opens 4 windows on the screen (see figure 2) in the case of vanilla X, one window (see figure 3) in the case using TCL/TK:

- **STORAGE:** In this window the event-sink parameter is set. In the initialize-phase of a run the storage parameters can be changed in this window. Furthermore, this window shows the name of the *recorder* process, as known by runcontrol. This name can be changed by typing "set recorder = `newname`" in the **COMMAND LINE** window.

- **RUNPAR:** This window displays the run parameters. Some of the run-parameters (run number, maximal number of events, event source) can be modified during the initialization phase of a run. Furthermore, this window shows the name of the *collector* process, as known by runcontrol. This name can be changed by typing "set collector = `newname`" in the **COMMAND LINE** window.

- **RUNCONTROL:** This window consists of a set of buttons that can be used to send commands to run control. Furthermore, this window displays if the operator is privileged to send commands. If the operator is not privileged clicking on the buttons has no effect. Furthermore, this window shows the name of the *runcontrol* process, as known by operator. This name can be changed by typing "set runcontrol = `newname`" in the **COMMAND LINE** window.

- **COMMAND LINE:** This line can be used to send commands to *runcontrol*. Commands are only sent to *runcontrol* if the operator is privileged. To enter the privileged mode type PRIVI. To leave this mode type NOPRIVI. The command EXIT stops the *operator* process.

The *operator* process can be started and stopped at any time. This does not influence the state of the online system or a run. However, the *operator* is the interface between the person on shift and the online system. Stopping the *operator* means that the person on shift has no knowledge on the state of the online system other than that garnered by browsing through the messages of the logbook/murmur windows (as described below) or by "ask"ing a process for the value of its user or system variables.

<u>Operator User-level Commands:</u>

QUERY_RUNPAR, QUERY_STORAGE, and PROMPT.

<u>Operator User-level Variables:</u>

runcontrol.

<u>Operator Critical Variables:</u>

runcontrol.

3.7 a. <u>User Level Variables</u>

<*runcontrol*>: String. Initial value is "runco". This variable determines which *runcontrol* process is in use.

### 3.7 b. Using the Operator Process

As mentioned before, when starting the *operator*, four screens (figure 2) or one screen for TCL/TK (figure 3) appear on the workstation display. First of all, the *operator* process needs to be privileged to send messages to *runcontrol*. This can be achieved by typing the command

PRIVI

in the Run Control Commands window, after the "command>>" prompt.

Afterwards, this window can be used to send commands to *runcontrol* by simply typing the command. An exception to this is the command 'EXIT', which will be passed to the *operator* process causing it to stop and not to *runcontrol*. See the Technical Guide for a summary of the commands understood by *runcontrol*.

The Run Control window contains seven buttons, together with a notification which displays if the person on shift can send messages to *runcontrol* or not (privileged/NOT privileged). The remainder of this section contains a description of the default actions performed when clicking on the buttons. The actions taken depend on the input file of *runcontrol*, `command.list`, and will change from this description if the file is modified. The detailed actions of *runcontrol* to the commands listed below are summarized in the Technical Guide.

1. **Start cold**– When the run is in a stopped state (status Idle), the 'BEGIN_RUN_COLD' command is sent to *runcontrol*.
2. **Start warm**– When the run is in a stopped state (status Idle), the 'BEGIN_RUN_WARM' command is sent to *runcontrol*.
3. **Stop** – When the run has not yet stopped (status not Idle), it sends command 'STOP_RUN' to *runcontrol*.
4. **Suspend** – When a run is in progress (status Run) it sends command 'SUSPEND_RUN' to *runcontrol*.
5. **Continue** – When the run has been suspended (status Pause), it sends command 'CONTINUE_RUN' to *runcontrol*.
6. **Download** – When the run is stopped (status Idle) it sends command 'DOWNLOAD' to *runcontrol*.
7. **Initialize** – When the run is stopped (status Idle) it sends command 'INITIALIZE' to *runcontrol*.

### 3.7 c. Modifying Run Parameters

When sending an 'INITIALIZE' or 'BEGIN_RUN_COLD' command to *runcontrol*, *runcontrol* asks the person on shift to confirm the run (and storage) parameters. The 'Run Control Commands' window now displays a button in the upper right corner. One should click on this button if the run parameters are set properly. The run parameters that can be modified are the only parameters which are shown at this time in the 'Run Parameters' window. Both the 'INITIALIZE' and 'BEGIN_RUN_COLD' commands will increment the run number (unless the run number is zero) before asking new information of the person on shift. 'BEGIN_RUN_WARM' only increments the run number.

After clicking the button, the run parameters are sent to *runcontrol*, and a new button appears in the 'Run Control Commands' window. This time it is possible to change the Storage parameters in the 'Storage' window. One can put "dummy" in the event sink, in which case no output will be written. Or one can put a directory name in this window, which means that the *recorder* will write data to that specific directory. The default is to write in the working directory from which *recorder* was started. 'BEGIN_RUN_WARM' performs none of these actions but simply re-uses previous settings.

Both 'BEGIN_RUN_WARM' and 'BEGIN_RUN_COLD' will ask the person on shift for a begin run comment. This is to be typed in at the command line. *Runcontrol* passes this comment to the *collector* process as part of the BEGIN command processing.

After clicking the button, the storage parameters will be sent to *runcontrol*, and all parameters are modified (or set) according to the needs of the *operator*.

### 3.8 XPC Process

The XPC (eXperimental Process Control) checks if all processes are alive and maintains the Common Table and Status Path of the (UNIX) processor on which it runs. (Re)starting and stopping processes can be done using the XPC. When the XPC starts, it first creates shared memory. All global variables are stored in this shared memory. If the XPC dies abnormally, this shared memory will be used by the next XPC-process. After that the XPC determines which processes are running. All online processes will be monitored, including processes which are not started by the XPC but rather in the normal UNIX fashions. Only processes that are started by the XPC can be restarted automatically.

XPC User-level Commands:

START, STOP, IGNORE, INTERRUPT, UPDAT_TBL and TEST_XPC.

XPC User-level Variables:

queuetest.

Process Critical Variables:

None.

#### 3.8 a. User-Level Commands

The descriptions of the *xpc* user-level commands are given below:

**START** <*[YES/NO] [-p t1] name[@node] [args]*>: Starts process *name* at node *node*. When the first argument is "YES", the process is restarted automatically after it dies abnormally. The "-p" argument determines the time between successive checks of the status of the process. The default value is 60 (seconds).

**STOP** <*process_name*>: If the process is running locally, the *xpc* stops the process by sending the "EXIT" command, otherwise this "STOP" command is forwarded to the remote *xpc*.

**IGNORE** *[ ON/OFF ]* <*process_name*>: Normally, the *xpc* checks the state of all local processes. With this command, the state of the corresponding process is not checked by the *xpc*. The use of this command is not recommended. If <process_name> is not supplied, it is assumed to be the sending process. If <process_name> IS supplied, then one of ON/OFF must also be supplied. ON indicates the process should be ignored by the XPC. OFF indicates the XPC should again pay attention to whether the process is responding. The default is ON, so the command IGNORE with no arguments is interpreted to mean the sending process should be ignored.

**INTERRUPT** <*process_name*>: Sends a signal to a process, causing that process to set the variable "interrupt_flag" to 1. It is recommended that users programming time consuming routines check regularly if this variable is set (see the discussion of this variable at the end of Chapter 5).

**UPDAT_TBL** <*process_name node status*>: This command is used among different *xpc*'s to notify each other that the common table needs updating.

**TEST_XPC** : Upon receiving this message, the *xpc* sends an "ALIVE" message back to the sending process, which is expected to be the *xpc_checker*. This message is used by the *xpc_checker*, and is for internal use only.

3.8 b. <u>User Level Variables</u>

<*queuetest*>: Integer. This variable is used by the *xpc* to periodically test the status of message queues. This variable is for internal use only, and should not be modified by other processes.

## 3.9 XPC Checker

There is only one *xpc_checker* running on UNIX in the complete system. It checks all machines where an XPC should be running (An XPC should be running if it has been started once after the latest Reset). If an XPC should be running on a node and is found to be dead, a new XPC is started on that node. The checker itself is checked by the XPC on the node where it is running and might be restarted if it is dead.

<u>XPC checker User-level Commands:</u>

ALIVE

<u>XPC checker User-level Variables:</u>

None.

<u>XPC checker Critical Variables:</u>

None.

3.9 a. <u>User-Level Commands</u>

The descriptions of the *xpc checker* user-level commands are given below:

**ALIVE**: This command message is the response of an *xpc* process to a TEST_XPC message from the *xpc_checker*.

## 3.10 Logbook Process

The *logbook* process is essentially a means of maintaining a history of UNIDAQ operations. Starting and stopping of processes is recorded along with run statistics. All the processes in the $TOPUNIDAQ/src/control directory have a number of error codes assigned to them which are used to track their operation, and thereby the operation of UNIDAQ. The *logbook* process does not run on VXWORKS machines.

The *logbook* is able to receive messages from all other processes as well. In general the messages consist of an error-code combined with from zero to four strings. If the error-code is zero, the strings are simply logged to a file called log.bookYYMMDD, where YYMMDD is the current date and sent as-is for display by murmur. If the error code is non-zero, then it is assumed to be a pre-defined error code and the strings are assumed to be the parameters of the associated message text. Known error codes and text are read at process initialization and have severities ranging from Success and Informative to Warning and Error. The severity is used to route the message to the correct murmur window. Changes to the error code/text are propagated from the murmur-server via the murmur_con program, but do not require recompiling *logbook*. In all cases messages are both logged to disk and sent to the murmur-server. By default at installation time the log.bookYYMMDD files are stored in $TOPUNIDAQ/log, but this directory can be changed during the installation. See the Technical Guide and the Installation Guide for further discussions of the error codes and messages.

Logbook User-level Commands:

read, log and RoundUp

Logbook User-level Variables:

None.

Logbook Critical Variables:

None.

### 3.10 a. Logbook Commands

*Logbook* understands the following commands:

**read** <*file*>: The named text <*file*> is read and copied to the log.bookYYMMDD file. <*File*> must either be in the working directory from which *logbook* was started or be an absolute path name.

**log** <*message*>: The specified <*message*> is copied into the log.bookYYMMDD file. These messages also appear in the murmur "General Log" window. This command is equivalent to the RoundUp command, with an <*error code*> of zero and <*npar*> = 1.

**RoundUp** <*error code*> <*npar*> [<*n message 1*>] ... [<*n message npar*>]: This command, which is not accessible at the shell command line due to the embedded new-line characters, is that used to log murmur messages. <*error code*> is the pre-set murmur error code, <*npar*> is the number of string parameters to follow in the range 0–4, and the remainder of the parameters are the optional, new-line prefixed, string parameters with the number supplied matching the value of <*npar*>.

### 3.10 b. Viewing the Logbook Files

If murmur is running, the progress of UNIDAQ is reported in the X-windows owned by the murmur server (see the next chapter). Alternatively, or if the murmur server is not running, the *logview* tool may be used to view the current log.bookYYMMDD file. See the "Additional Tools" chapter of this manual for details.

## 3.11 UNIwish shell

If TCL/TK is installed, the *UNIwish* shell is created as an interface between TCL/TK and UNIDAQ. *UNIwish* behaves as a standard UNIDAQ process, while having the full functionality of the "wish"-shell, which is distributed with the TCL/TK package. Below, *UNIwish* will first be described as a standard UNIDAQ process, followed by a description of differences between *UNIwish* and wish.

### 3.11 a. UNIwish as a UNIDAQ process

*UNIwish* is able to receive standard UNIDAQ commands. Furthermore, it can receive NOVA-buffers, depending on the tcl-script used when starting this process. Next to checking for messages and NOVA-buffers, *UNIwish* is also able to react to commands given by the user, either using the mouse or the keyboard.

UNIwish User-level Commands:

none.

UNIwish User-level Variables:

None.

UNIwish Critical Variables:

None.

### 3.11 b. Differences between wish and UNIwish

*UNIwish* responds to all UNIDAQ commands and command line options. It also registers itself in the UNIDAQ Common Table and Status Path. Furthermore, the wish "exit" command has been modified to nicely clean up the Common Table and Status Path. Next to these UNIDAQ neccesities, differences are kept to a minimum. There are some additional tcl-commands and an additional global variable can be used in the tcl-scripts. All of this is described below.

Additional UNIwish global variables

**ownpname:** Contains the name by which this invocation of UNIwish is known by the other UNIDAQ processes.

Additional UNIwish commands

**ask** < *process_name* > < *message* >**:** Sends a message to process *process_name*. Returns a tcl-error when the message cannot be sent.

**proc_num_var** < *process_name* >**:** Retrieve the total number of variables stored in the status path for process *process_name*.

**proc_var_num** < *process_name* > < *variable number* > **:** Retrieve the value of variable *variable number* of process *process_name* from the status path. *Variable number* must be less than or equal to the total number of variables of the process held in the status path.

**proc_var** < *process_name* > < *variable_name* >**:** Gets the value of the variable *variable_name* belonging to process *process_name* using the Status Path. Returns the error code "Error in read_value" when the call fails.

**date_val** < *date* > **:** Converts an integer value to a date, using the standard "ctime" subroutine. This allows for displaying the "start-of-run" and "end-of-run" dates as stored by the collector process.

**nova_get :** When first called it connects to the NOVA buffer manager, and gets the first buffer. Subsequent calls only get new NOVA buffers. Returns "0" when succesfull, otherwise it returns a "-1".

**nova_put :** Releases a NOVA buffer.

**next_event :** Gets the next event out of the current NOVA buffer. Returns "0" when succesfull, otherwise it returns a "-1". If this call is not succesful, the NOVA buffer has to be released and a new buffer has to be retrieved from NOVA.

**event_data** < *offset* > < *word_type* > **:** Retrieve a dataword from the event. The word_type can be "integer", "real", or "string". The offset starts at 0.

**event_header** < *offset* > **:** Gets a header word from the current event. The offset starts at 0.

**status_num_var :** Retrieve the total number of process entries in the status path.

**status_var_info** < *status_number* > **:** Retrieve the status path information of the process in position *status_number* within the table.

## 3.12 UNIview

*UNIview* is a tool allowing a graphical view of the UNIDAQ processes. It is implemented as a TCL/TK script and therefore is only available when UNIDAQ is built to use TCL/TK. It presents a window to the user showing the process tree of UNIDAQ, with machines shown across the top of the view, and the processes running on those machines shown as downwards branches. The process boxes are color coded, with red showing dead processes, yellow showing busy processes and green showing live processes, where the dead, busy and live states are determined from the information in the UNIDAQ process status table.

Clicking the left mouse button on any process icon pops up a box showing the list of system and user variables of the process together with their current value. Clicking the "EXIT" button on the expanded process closes it back to its iconic state. UNIDAQ machines are normally shown expanded, but can be compressed to icons (with the process branches now hidden) by clicking on them.

If more than six UNIDAQ processes are found in the common table the additional processes will be hidden off the side of the display. A scrollbar at the bottom of the display will bring these processes into the active window.

The UNIview display does not update regularly, but rather gives a snapshot of the UNIDAQ state. An "UPDATE" button exists on the display though which will query the status path and common table, and revise the display with the new information.

# 4. MURMUR

Murmur is a client-server, error reporting package developed at Fermilab. Errors have a VMS-like syntax, with a program related prefix, a severity, a short mnemonic, and a more complete text.

If the murmur server is running it will have three associated display windows. The first is labeled "General Log" and contains all purely text messages (such as those from the "ask logbook log `some text`" shell command) and those with either Success or Informative severity. Messages in these severity categories track the state of UNIDAQ, eg, starting or ending runs.

The second window is labeled "Warning messages" and contains messages of that severity. For example if you are running *operator* but are not privileged, and attempt to begin a run, you will be notified here of your failure to do so.

The final window is labeled "!! FAIL !!" and contains messages of Fatal severity. For example the premature death of a program is recorded here.

## 4.1 Starting the Murmur Server

The murmur server will only run on a SUN Sparc or an SGI platform. If one is not available, that is not a disaster – UNIDAQ does not require murmur, rather murmur is a convenient tool for displaying the logged system status.

To start the murmur server, add to the "Start_node" script on the appropriate machine the line

murmur_Start

Since murmur does not register with the common table, this line can go anywhere in the script, with one restriction: murmur must start before the *logbook* process. This is because *logbook* is a client of the murmur server.

The alternative to the "Start_node" script for starting the murmur server is to simply log into the node, and then issue the command

murmur_Start

at the shell prompt.

### 4.2 Stopping the Murmur Server

To stop the murmur server, log into the node where it is running and issue the command

murmur_Stop

It is best, but not required, to do this **after** the *logbook* process is stopped.

## 5. AN EXAMPLE SESSION: COSMIC RAY TESTS OF SDC PROTOTYPE MUON DRIFT TUBES

In this chapter we provide a complete example session with parts of the UNIDAQ Portable DAQ System. This example session is based on the use of the system in *Cosmic Ray Tests of SDC Prototype Muon Drift Tubes.* The example demonstrates how to modify *collector* code, according to the current configuration of the hardware and the data source, how to add new histograms to the dynamically linked analyzer process *ana_d*, display variables with dataview, and play back data files for additional analysis.

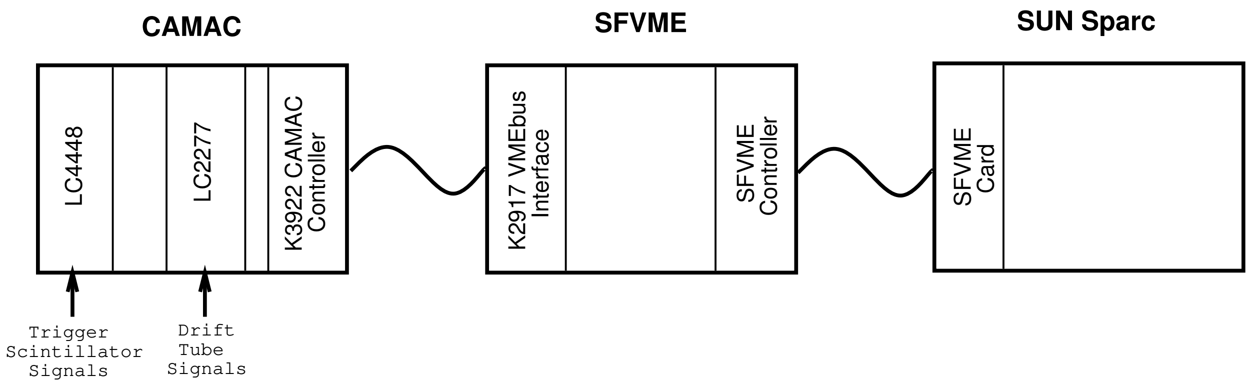


Figure 4. Hardware Configuration of Cosmic Ray Tests of SDC Prototype Muon Drift Tubes

### 5.1 Hardware Configuration of the Test and Modifying the Collector

*Collector* interacts with the event source, reads the event data from the event source, and stores the results into a buffer. The operations related to the event source are hardware dependent, and should be provided by the user of the system. In this section, we demonstrate how to attach these routines to the system. We start with the hardware configuration of the example system. The hardware configuration of the Cosmic Ray Test is illustrated in Figure 4.

The user is supposed to provide seven procedures to the *collector*, one for process_event to call, one each for program initialization and termination, and one for each of the following *collector* commands: BEGIN, PAUSE, RESUME, and END. These routines are respectively called by the generic names user_event, user_init, user_exit, user_begin, user_pause, user_resume and user_end and appear in the file user_event.c . The content of each of the user procedures in this test, based on the skeleton provided in the UNIDAQ distribution, is given in Figure 5. "Mode" and "run_type" are the *collector* user-level variables 'mode' and 'runtype'. "Max_length" is the maximum size the event may attain (in words), and "buffer" is the array of length "length" into which the data is to be placed. "Event_length" and "event_ID" are returned as the number of words filled in buffer by the data acquisition sequence and the buffer type. Note that the calling code for PSEUDO events, typically events filled with random numbers, is also included in this example.

---

```
#include "camlib.h"
#define L4448 7 /* Slot number of L4448 */
#define L2277 12 /* Slot number of 2277 */
#define MAX_EVENT_SIZE 100  /* used for pseudo event initialization */

user_init ( max_length )
int *max_length;
{
    if (strcmp(eventsource,"PSEUDO") == 0) {
        eventSrcType = PSEUDO;
        *max_length = MAX_EVENT_SIZE;
    }
    else {
        *max_length = 1000;
        eventSrcType = OTHER;
    }
    return;
}

user_event (buffer, length, event_length, event_ID, max_length, mode)
int *buffer;        /* pointer to buffer                          */
int *length;        /* buffer length                              */
int *event_length;  /* user_event must set the event length       */
int *event_ID;      /* user_event must set the event ID word      */
int *max_length;    /* max words an event should occupy           */
int *mode;          /* mode of event taking, for those who use it */
{
    int dat, q, x, i;
    int mask, status;

    if (*length < *max_length) return ERANGE;

    *event_length = 100;
    status = CAM_WaitLAM(3);
    CAMAC(NAF(L4448,0,9), &dat, &q, &x);   /* Clear LAM */
```

```
    if (status == ETIMEDOUT)  {            /* timedout */
        return status;
    }
    if (*mode == 0)  {
        *event_ID = EVENT_RECORD;
        CAMAC(NAF(L4448,0,11), &dat, &q, &x);
        CAMAC(NAF(L2277,0,0), &dat, &q, &x); /* Read TDC */
        i=0;
        while (q !=0) {
           buf[++i] = dat;
           CAMAC(NAF(L2277,0,0), &dat, &q, &x); /* Read TDC */
        }
        buf[0] = i+1;                             /* Store number of TDC "hits"  */
        CAMAC(NAF(L2277,0,9), &dat, &q, &x);
        *event_length = i+1;
    }
    else  {
        *event_ID = ENVIRONMENT_RECORD;
        *event_length=100;
        flush_buffer("","");
    }
    return(0);
}
```

**user_begin(run_type, max_length)**

```
int *run_type, *max_length;
{
    if (strcmp(eventsource,"PSEUDO") == 0)  eventSrcType = PSEUDO;
    else                                    eventSrcType = OTHER;

/*
*  eventSrcType may have changed, so be sure to reset max_length
*/
    if (eventSrcType == PSEUDO)  {
         *max_length = 150;
         pseudo_begin();
         return 0;
    }
    else {
        *max_length = 1000;
        CAMOPN();
        CSETBR(1);
        CSETCR(0);
        CGENC();
        CGENZ();
        CREMI();
        CAM_DisableLAM();
        return;
    }
}
```

```
user_pause (mode)
int *mode;
{
    int dat, q, x;

    if (eventSrcType == PSEUDO) {
        pseudo_pause();
        return 0;
    }
    else {
        CAMAC(NAF(L4448,0,24), &dat, &q, &x);   /* Disable LAM */
        CAM_DisableLAM();
        return;
    }
}

user_resume (mode)
int *mode;
{
    int mask, dat, q, x;

    if (eventSrcType == PSEUDO) {
        pseudo_resume();
        return 0;
    }
    else {
        mask = 1 << (L4448 - 1);
        CAM_EnableLAM(mask);
        CAMAC(NAF(L4448,0,26), &dat, &q, &x);   /* Enable LAM */
        dat = 12161;
        CAMAC(NAF(L2277,0,7), &dat, &q, &x);
        return;
    }
}

user_end (run_type)
int *run_type;
{
    if (eventSrcType == PSEUDO) {
        pseudo_end();
        return 0;
    }
    else {
        CAM_DisableLAM();
        CAM_Close();
        return;
    }
}
```

```
user_exit() {
    return;
}
```

Figure 5. Procedures corresponding to the collector camac operations.*

---

The user is supposed to set the event source after starting the *collector*. Two event sources (CAMAC and PSEUDO) are provided in UNIDAQ 2.3 . CAMAC in this context means any real set of hardware, including VME and CAMAC modules. If the user wants to use the hardware setup given in Figure 4, he/she should set the event source as follows:

ask collector set event_source = CAMAC.

Otherwise, if the user wants to test the system with a pseudo-random number generator, he/she should set the event_source to PSEUDO:

ask collector set event_source = PSEUDO.

If the *operator* process is in use, then instead of the 'ask' tool one simply enters CAMAC or PSEUDO in the appropriate window box when starting a run.

See the Technical Guide for details of other subroutines available within the *collector* process.

## 5.2 Histograms and the Analyzer

The dynamic analyzer process *ana_d* is used to plot on-line histograms of the incoming data. The histogram filling procedures are dynamically linked to the *ana_d*. In our example, four histograms are defined for number of hits, drift time leading edge, drift time tailing edge, and width. These histograms are filled using the procedure FILL_MUON which is defined in muon_hist.f (Figure 6). Two arguments are passed to the routine, the event data (array IDATA) and the event header (array IHEADER).

---

```
SUBROUTINE  FILL_MUON(IDATA, IHEADER)
INTEGER IDATA(*), IHEADER(*)
COMMON/PAWC/HMEMORY (10000)
LOGICAL FLAG/.TRUE./

IF (FLAG) THEN
    CALL HBOOK1(1, 'Number of Hits $', 80, 0, 80, 0.)
    CALL HBOOK1(2, 'Drift Time Leading Edge $', 170, 0, 3400, 0.)
    CALL HBOOK1(3, 'Drift Time Tailing Edge $', 170, 0, 3400, 0.)
    CALL HBOOK1(4, 'Width', 50, 0, 100, 0.)
    FLAG = .FALSE.
ENDIF

IEvent_Muon = IDATA(1)
CALL HF1(1, FLOAT(IEvent_Muon), 1.)
```

* See *Usage Guide of CAMAC Library for UNIX Version 0.8*, Y. Yasu, KEK Online Group, for details of these CAMAC subroutines.

```
DO III = 2, IEvent_Muon
    IHit_Muon = RSHIFT(AND(IDATA(III), x'10000'),16)
    IChannel_Muon = RSHIFT(AND(IDATA(III), x'3E0000), 17)
    ICount_Muon = AND(IDATA(III), x'FFFF)
    IF (IHit_Muon.EQ.1) THEN
       CALL HF1(1,FLOAT(ICount_Muon), 1.)
       Lead_Channel = IChannel_Muon
       Lead_Count = ICount_Muon
    ELSE
       CALL HF1(3, FLOAT(ICount_Muon), 1.)
       IF (Lead_Channel .EQ.IChannel_Muon) THEN
         CALL HF1(4, FLOAT(Lead_Count - ICount_Muon), 1.)
       ENDIF
       Lead_Count = 0
       Lead_Channel = 0
    ENDIF
ENDDO
RETURN
END
```

Figure 6. Filling the Histograms.

---

In order to display these histograms, the user should send the following commands to the *ana_d*:

```
link muon_hist.f
add muon_hist.f fill_muon
display muon_disp 5
```

Here, a muon_disp.kumac is defined as shown below. The value "5" supplied on the associated "display" command is the time interval in seconds between successive executions of the kumac. Any convenient interval could have been chosen instead. Upon first entry to the macro, "FIRST" is set so that the display is properly initialized and the histograms properly plotted.

```
MACRO muon_disp FIRST = NO
IF [FIRST] = NO GOTO UPDATE
   zone 2 2
   histogram/plot 1 k
   histogram/plot 2 k
   histogram/plot 3 k
   histogram/plot 4 k
update:
   histogram/plot 1 u
   histogram/plot 2 u
   histogram/plot 3 u
   histogram/plot 4 u
   call igterm
   return
```

## 5.3 Displaying Data with Dataview

### 5.3 a. Without TCL/TK

In order to display data using the *dataview* process two files have to be edited. Both these files are located in directory $(USERUNIDAQ)/dataview. The first file to be modified is file datawin.h, which contains the description of the dataview window. As an example, to display only the number of hits, which is stored in the first data word (see Figure 5), this file would look like:

```
struct dw_def {
    char txt[20];
    int(*conversion)();
};
extern int nr_hits();
struct dw_def dview[] = {
    "nr hits",(int(*)())nr_hits,
};
#define MAX_N_SL_DATA 10
int n_sl_data = sizeof(dview)/sizeof(struct dw_def);
```

The `nr_hits()` subroutine creates a character string from the data. This routine needs to be coded in dataconv.c, the code would look something like:

```
nr_hits(header,data,txt)
int *header,*data;
char *txt;
{
    if(header[1] == EVENT_RECORD) sprintf(txt,"%d",data[0]);
    else strcpy(txt,"No dataword");
}
```

After recompiling and starting the dataview process, a window will appear which contains the number of hits in the current event, and which continuously updates with each new event buffer.

### 5.3 b. With TCL/TK

As mentioned before, when using TCL/TK the *dataview* functionality is determined by a tcl-script called `dataview.tcl` located in the `$USERUNIDAQ/conf` directory and read by the *UNIwish* shell.

If one wants to display the number of hits (the first dataword), as in the previous example, this file needs to be modified in several places as shown below:

```
..
..
#-------------------------------------------------------------
# Initialize variables; add whatever you wish to display here
#---------------------------------------------------------------
```

```
set nrhits " Number of hits:"
set dtcolor white
..
..
#-------------------------------------------------------------------
# Add a label for each individual parameter (you might consider one for
# the description as wel)
#--------------------------------------------------------------------
frame .evtdata -borderwidth 1 -background $dtcolor
label .evtdata.nrhit -textvariable nrhits -anchor w -background $dtcolor \
-borderwidth 1 -relief raised -foreground black
..
..
#--------------------------------------------------------------------
# place it all in the window
#--------------------------------------------------------------------
place .evtdata.nrhit -relx 0.    -rely 0.  -relwidth .4 -relheight .2
..
..
#--------------------------------------------------------
# The events are read in, now fill the display variables
# with whatever you like checking on a lot or not at all
#-------------------------------------------------------
proc display_data {}{
global nrhits
set nrhits " Number of hits:  [event_data 0]"
}
..
..
```

After editing the file, dataview can immediately be restarted without any other intermediate action.

## 5.4 An Example Run of the Portable DAQ System

In this section, we demonstrate an example run, using a UNIX machine, of the portable data acquisition system, UNIDAQ 2.3 . Even though the system can be run in several different ways, the following set of commands are provided as an example to the user who does not have any experience with the system and wants to see it work at the most basic level. In this example, among the UNIDAQ user processes only *analyzer*, *dataview*, and *collector* are executed. The dynamically linked *ana_d* is that used in this example.

First, change directory to the one in which you have prepared `muon_hist.f` and `muon_disp.kumac`.

[1] cd my_directory

Then execute NOVA, the buffer manager process, with the following command.

[2] novad /tmp/nova &

Next start the *collector*, *dataview*, and *analyzer*.

[3] collector &

[4] dataview &

Steps [2] through [4] could as easily have been performed by simply typing the command "Start". In this case the Start_node script for your machine should have been edited to start only the NOVA, collector, and dataview processes, in addition to the standard XPC and *msg_server* processes.

Note: The dynamically linked *analyzer* process should always be run in the foreground, so start *analyzer* in its own window.

[5] ana_d

To see which processes are successfully started use the status tool.

[6] status

Next, link the files to paw in the *analyzer* interface with the following commands.

paw > link muon_hist.f

paw > add muon_hist.f fill_muon

To start the *collector* data processing, issue the following commands to collector:

[7] ask collector set eventsource = PSEUDO

Note: When you set the event source to PSEUDO the *collector* generates random data between zero and one hundred.

[8] ask collector begin

[9] ask collector resume

Note: Steps [7] through [9] can as well be handled by the *operator* process in conjunction with *runcontrol*.

And finally, in the *analyzer* interface use the display command to see the histograms plotted once every five seconds.

paw > display muon_disp 5

To end a run.

[10] ask collector end

[11] ask collector exit

[12] ask dataview exit

paw > display off

paw > unlink muon_hist.f

paw > exit

Then issue the Reset command to reset the system.

### 5.4 a. Interrupting a Process

If a process such as *collector* is going to take a long time to process a command or perform some action, it is a good idea to periodically check from within the code to see if the process has been told to abort. This is done by checking the value of a variable which all template-using processes hold in a global area. The variable is called `interrupt_flag`, and its value is normally zero. If this value is anything else, then the process has been told to abort its action, and it should immediately do so and return from the subroutine.

Doing this is not required, it is merely a convenient means of kicking some response back into a process. The variable value is always reset to zero by the main loop of the template.

## 5.5 Playback of a Data File

### 5.5 a. Within the UNIDAQ environment

It is possible, once a data file has been recorded on disk, to play it back through UNIDAQ again. This is useful, for example, if you wish to run *analyzer* over again on a data set. Several methods are available for selection. In this example we choose to use the *from_pipe* tool. See Chapter 6 for other methods. Using *from_pipe* the syntax is to issue the command

```
cat name-of-data-file | from_pipe
```

In this case *from_pipe* is the event source for NOVA instead of *collector*. Also, you may want to force your *analyzer* to have a sufficient priority (-p switch) to receive all events in this case.

### 5.5 b. Playback without the UNIDAQ environment

Both the *ana_d* and *ana_nd* processes have off-line equivalents, namely *ana_d_off* and *ana_nd_off*. These processes can be run independently of the rest of the UNIDAQ system. To do so, simply start them as in the previous section, and after you have set up the program as you wish, issue the `analyze` command at the PAW> prompt:

```
analyze name-of-data-file
```

The specified data file will be played back through the process with analysis as specified by the linked subroutines.

# 6. PASSING EVENT BUFFERS BETWEEN MACHINES IN A DISTRIBUTED ENVIRONMENT

## 6.1 Using the Buffer Manager to Pass Data

With the incorporation of NOVA v1.05, a natural means now exists to pass data buffers between machines independent of byte ordering. The means of doing this is quite simple. On the machine where the *novad* process is running, and which is the source of event data, a network daemon process must also be started. The initial 'start_node' script starts this automatically after *novad* is started. To do it manually, you must type the command

```
novand >& /dev/null &
```

Following this, processes such as *analyzer* which operate on event data buffers may be started on other machines by adding "-m nova_machine" to the command line, where "nova_machine" is that one on which *novad* and *novand* are running.

On some machines, warning messages are printed in the window where this *analyzer* process is running, but these can be ignored; they are merely informative.

Note that if you are using the global section *analyzer* process *ana_gl*, then the histograms created are available only on the machine on which it is running. To access them, PAW must also be run on that machine.

A special process called *receiver* is available which will fetch events from other machines and then place them in a local NOVA buffer. See below for a description of this method.

## 6.2 Using Pipes to Pass Data

UNIDAQ 2.3 is developed to run on multiple workstations. Using the method of the previous section buffers of events can be passed to machines other than that on which they were acquired. A second method using pipes is also provided to be able to pass event buffers between machines. For example, the user may want to run the *collector* in workstation W1 and the *analyzer* in workstation W2. The pipe connection is established by means of two processes: *to_pipe* and *from_pipe. To_pipe* gets the event buffer from NOVA and writes the buffer to its stdout, and *from_pipe* reads the data from its stdin and places it into a NOVA buffer, as illustrated in Figure 7. Using this method NOVA must be running on both involved machines.

An example run on two different workstations is demonstrated below. In this example, it is assumed that the *collector* process is running in Workstation 1, and the *analyzer* is running in Workstation 2.

| <u>WORKSTATION 1</u> | <u>WORKSTATION 2</u> |
|---|---|
| [1] novad /tmp/nova & | [4] novad /tmp/nova & |

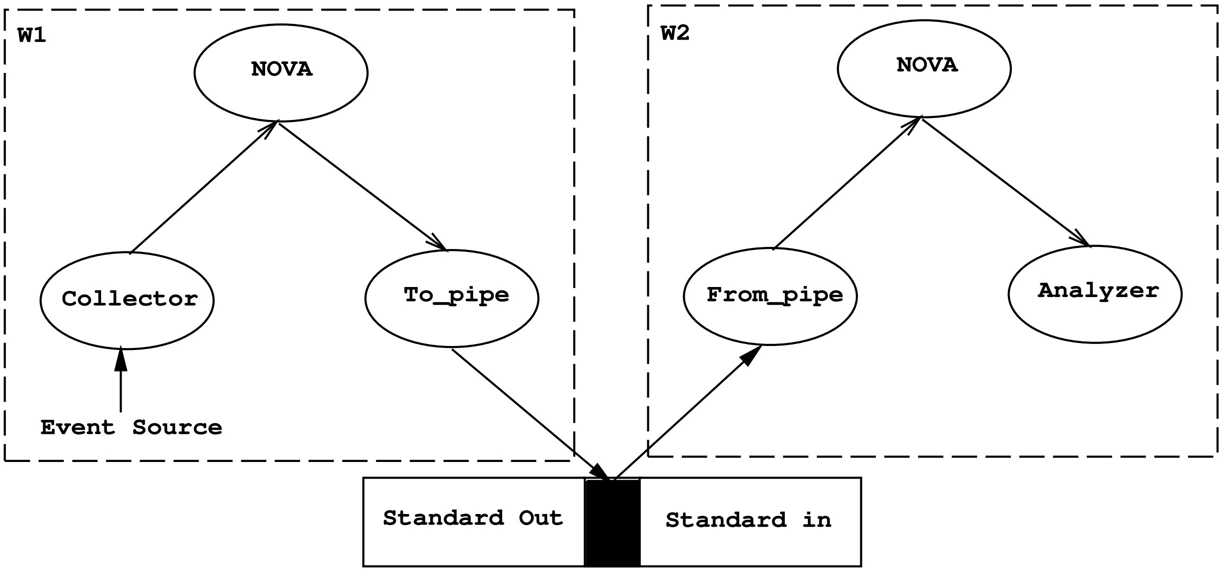


Figure 7. Running Analyzer on a different workstation using pipes.

[2] collector &
[3] ask collector set eventsource=PSEUDO

[5] to_pipe | rsh [workstation2] \
'$(TOPUNIDAQ)/bin/from_pipe'
[7] ask collector begin
[8] ask collector resume

Next add any other processes, ie, analyzer, recorder, dataview, UNIdump, ...
[6] analyzer

Note the rsh UNIX command requires a .rhosts file to be created in the user's home directory on Workstation 2. On the HP one must use 'remsh' instead of 'rsh'. On all machines the actual directory must be specified because $(TOPUNIDAQ) will not be known in the rsh environment.

It is apparent that *from_pipe* and *to_pipe* can also be used in a single workstation together with UNIX shell commands for some other purposes. For example, the following command can be used to get the octal dump of event data:

to_pipe | od -h | more

### 6.3 Using *receiver* to Pass Data

A third method of passing events between machines is also available. The *receiver* process is designed to fetch NOVA buffers from a remote machine, and put this data into a local NOVA buffer, from which the data can be passed to other processes. In order for this to work, the remote machine must run both *novad* and *novand*. The local machine must run *novad*, and *receiver* must be started using the command

`receiver -m remote_machine >& /dev/null &` (UNIX)
`sp receiver, ''-m'', ''remote_machine''` (VXWORKS)

Other processes can be started on the local machines in their normal way, that is to say without the -m option.

### 6.4 Using UNIDAQ in a combined UNIX-VXWORKS environment

In a combined UNIX-VXWORKS environment, the control processes will run on the UNIX workstation, while datataking will be done on VXWORKS. The data can be recorded either on VXWORKS, or on the UNIX workstation. *Analyzer* and *dataview* processes are only available on the UNIX machine. In the example below it is assumed that *recorder* is located on the UNIX workstation.

1. The nodefiles on both UNIX and VXWORKS need to be modified to contain the internet addresses (name.network) of both machines.
2. Both Start-scripts need to be modified as shown below to load and start all needed processes.

The Start_unix file should look as follows:

```
#!/bin/csh
onintr -
source $USERUNIDAQ/setup.csh
msg_server >& /dev/null &
xpc >& /dev/null  &
if ( !  $?LOCAL_NOVA ) then
   setenv LOCAL_NOVA /tmp/nova
endif
if ( -e $LOCAL_NOVA) rm $LOCAL_NOVA
novad $LOCAL_NOVA  >& /dev/null  &
sleep 1
ask xpc START receiver -m <vxworks_node-name>
ask xpc START recorder
ask xpc START dataview
ask xpc START YES runco
ask xpc START YES logbook
ask xpc START YES xpc_checker
```

The Start_vxworks file should contain the following:

```
cd lib
ld <vxshm.o
cd ../NOVA/src
ld <novad
cd ../netsrc
ld <novand
cd ../..
sp novad_main
sp novand_main
cd bin
ld <msg_server
sp msg_server
```

```
ld <collector
sp collector
ld <status
ld <repair
ld <ask
```

3. In one window rlogin to VXWORKS
4. On UNIX type `Start_local`, and on VXWORKS type `<Start_vxworks`
5. On UNIX start *operator* and *analyzer* as detailed in previous sections. Start the run using *operator* as explained earlier.

Afterwards, you can stop UNIDAQ on the UNIX machine by typing `Reset_local`. On VXWORKS you have to stop each individual process, using the *ask* tool.

# 7. ADDITIONAL TOOLS

Of the tools described below, only the *status*, *ask* and *repair* tools are available on VXWORKS platforms. Differences between their useage in UNIX and VXWORKS are described in the corresponding sections.

## 7.1 UNIdump

The UNIdump tool allows the user to display the event data. For each event, the tool displays the event header, and then for each word of the event, it displays the word number, the integer representation, the first two bytes integer, the last two byte integer, the ASCII representation, and the hex representation of the four bytes. See figure 8 for an example of the UNIdump output.

UNIdump has three commands which are ON, OFF, and EXIT. The ON command allows the user to specify a range of words to be displayed for each event. For example ON 10 20 would display the header, and then the tenth word through the twentieth word of the event buffer. The default state, which is ON with no parameters, prints the entire content of the user data portion of the buffer, excluding the event header. The OFF command suspends all displays when issued. The EXIT command allows the user to exit the process from the system.

Event Length = 106 words

| | | | | | |
|---|---|---|---|---|---|
| Event Type = 0 | Run Number = 0 | | Event Number = 14 | | |
| Mode = 0 | Reserve One = 0 | | | | |
| Word Number | I*4 | I*2(1st) | I*(2nd) | ascii | hex |
| 0 | 106 | 0 | 106 | ...j | 0000006A |
| 1 | 0 | 0 | 0 | .... | 00000000 |
| 2 | 0 | 0 | 0 | .... | 00000000 |
| 3 | 14 | 0 | 14 | .... | 0000000E |
| 4 | 0 | 0 | 0 | .... | 00000000 |
| 5 | 0 | 0 | 0 | .... | 00000000 |
| 6 | 84 | 0 | 84 | ...T | 00000054 |
| 7 | 83 | 0 | 83 | ...S | 00000053 |
| 8 | 75 | 0 | 75 | ...K | 0000004B |
| 9 | 31 | 0 | 31 | .... | 0000001F |
| 10 | 13 | 0 | 13 | .... | 0000000D |
| 11 | 5 | 0 | 5 | .... | 00000005 |
| 12 | 46 | 0 | 46 | .... | 0000002E |
| 13 | 73 | 0 | 73 | ...I | 00000049 |
| 14 | 15 | 0 | 15 | .... | 0000000F |
| 15 | 66 | 0 | 66 | ...B | 00000042 |
| 16 | 13 | 0 | 13 | .... | 0000000D |
| 17 | 84 | 0 | 84 | ...T | 00000054 |
| 18 | 13 | 0 | 13 | .... | 0000000D |
| 19 | 25 | 0 | 25 | .... | 00000019 |
| 20 | 2 | 0 | 2 | .... | 00000002 |

Figure 8. Dump Output

## 7.2 Status

The status tool allows the user to view which processes are connected to the common table. The status tool displays the name of the process, the machine on which the process is running if different than the local node, the process's pid number and message queue number, and the status of the process either alive or dead. For remote processes the process-id and the message-queue id are not listed in the table. Message server processes and remote XPC processes do not show up in the common table. The table displays the knowledge the local node has on the complete system. When *msg_server* and XPC processes are all running this table usually gives an accurate picture of all processes running in the distributed UNIDAQ environment. See Figure 9 for an example of the status tool output.

| **process_name** | **node** | **pid** | **msgqid** | **status** |
|---|---|---|---|---|
| xpc | | 6310 | 78 | ALIVE |
| xpc_checker | | 6312 | 84 | ALIVE |
| collector | | 6314 | 2301 | BUSY |
| dump | | 6318 | 2152 | DEAD |
| recorder | | 6323 | 1403 | ALIVE |
| logbook | mhpbob | 0 | 0 | ALIVE |
| runco | | 6327 | 55 | ALIVE |
| from_pipe | | 6329 | 56 | ALIVE |
| operator | mhpbob | 0 | 0 | ALIVE |
| analyzer | | 6335 | 8 | ALIVE |

Figure 9. Status Output

The status-tool is available on both VXWORKS and UNIX. On UNIX it is activated by typing `status`, while on VXWORKS the status-tool first has to be loaded *e.g.*using the Start-script. Afterwards it is activated by typing `Status` (notice the capital "S" here).

## 7.3 Ask

Ask is a tool which allows the user to send messages to a process from the shell command line. The standard UNIX syntax for invoking a single message with ask is

ask process_name command

"Process_name" is the name of the process to which the command is directed, as stored in the common table. "Command" is one of the user-level or system-level commands recognized by the process, complete with all of that command's arguments. On VXWORKS this command would look like:

ask "process_name", "command"

For example, one could issue the command

ask collector set eventsource=PSEUDO (UNIX)
ask "collector", "set eventsource=PSEUDO" (VXWORKS)

from the shell command line.

A second use of ask allows multiple messages to be sent with a single invocation of the ask tool. In this form only the process_name only is supplied as argument to ask, eg, `ask collector`. Upon issuing this command the user will then get a "collector>" prompt, and several commands can then be sent to the *collector* process before exiting ask via QUIT, STOP or <CTRL-C>. For example,

ask collector
collector> set eventsource=PSEUDO
collector> begin
collector> STOP

NOTE: Do not leave this form of the "ask" tool by typing EXIT; this command will be sent to the process to which you are sending messages, causing it to stop itself.

## 7.4 Rask

The rask tool operates in the same fashion as the ask tool, except that it registers itself in the Common Table as a working process. In this way the results of commands such as "SHOW variable_name" will print results in the rask window instead of in the window in which the specified process started.

Multiple copies of rask may run at the same time. If a live process by the name rask already exists, it will pick a new name for itself.

The same restrictions on using the EXIT command as described for the ask tool apply to the rask tool.

## 7.5 Tell

Tell is merely a soft link to the rask tool.

## 7.6 Logview

The logview tool is used to display the last portion of the current log.bookYYMMDD file to the user's terminal screen. The program will then loop, waiting for 15 seconds and then checking the log file for modifications, and if any are found it will again display the last portion to the screen. Exit is via <CTRL-C>. This tool is a convenient means of monitoring the state of UNIDAQ in a terminal window, whether or not murmur is running, and only requiring that the *logbook* process is running.

If today's log file does not exist, the tail of yesterday's file will be printed once and the program will wait for today's file to appear. If neither file exists, the program exits with an error message.

## 7.7 CAMdo

The CAMdo tool can be used to perform single CAMAC operations. CAMdo is an interactive process which does not rely on or react to UNIDAQ message passing. When running CAMdo several questions need to be answered. CAMdo performs 24 bit data transfers. CAMdo is only available on HP, SUN and DECstations.

## 7.8 VMEdo

The VMEdo tool can be used to send single write and read commands to VME modules. VMEdo is an interactive process which does not rely on or react to UNIDAQ message passing. When running VMEdo several questions need to be answered, depending on the machine you are using. It provides the possibility of sending single and multiple read/write commands (eight or sixteen bits) to different sections in VME memory. VMEdo is only available on HP and DECstation platforms.

## 7.9 Uni_config

Using point and click methods uni_config allows you to edit files and create executables in such a way that UNIDAQ corresponds to your experimental needs. This process needs to be run in the foreground, since it requires the user to enter his/her favorite editor. Afterwards, a window pops up with several buttons (Nodefile, Start script, Runcontrol, Operator, Collector, Dataview, Exit). Clicking the Exit button will stop uni_config, clicking the Nodefile button will edit the nodefile, and clicking any of the other buttons will remove this window and pop up a new window from which the appropriate files can be edited. This new window also has an "exit" button; after clicking it the first window reappears. Clicking a button labeled "make" will create a new (modified) executable of the currently selected process in the $(USERUNIDAQ)/bin directory. The compilation result will show in the window where you started uni_config.

### 7.10 Startup

The startup tool does nothing further than create the common table. All UNIDAQ processes will do this automatically if the common table does not exist.

### 7.11 Shutdown

The shutdown tool does nothing more than delete the common table. This is done automatically during the various "Reset" procedures.

### 7.12 To_pipe

This tool is used to pipe NOVA buffers to its standard output. Such output can be used to pipe the buffers to other processes such as those on other machines. A sample usage is shown in Chapter 6 of this manual.

### 7.13 From_pipe

This tool is used to pipe NOVA buffers from its standard input into the NOVA buffer system, and the process is therefore a NOVA event source. Two sample uses of this tool are shown in Chapter 6 of this manual.

### 7.14 Ndmake

The *ndmake* tool is used when building the non-dynamically linked analyzer *ana_nd*. Type the command "ndmake -h" to see the syntax of its usage.

### 7.15 Example Scripts and Programs

Several sample scripts and programs are contained in the $USERUNIDAQ/examples directory as instructional examples. These are detailed here.

#### 7.15 a. example_script_a

This script runs NOVA, *collector*, *recorder*, *UNIdump*, and *status* to show how to talk to them, using random numbers as events, and looking at the buffers.

#### 7.15 b. example_script_b

This script runs NOVA, *collector*, *recorder*, *status*, and the dynamically linked *ana_d* showing how to dynamically link in and display histograms. Events contain random numbers.

#### 7.15 c. example

File example.c is a well commented, sample program exemplifying how to pick up and access NOVA buffers within the framework of the template, and how to make new user commands and variables. A working version of the program appears in the $TOPUNIDAQ/bin directory.

## 7.16 Repair

The repair tool allows the user to repair the table of process ids and message queue numbers (the Common Table) along with other associated shared memories when a process dies abnormally. It is suggested that a user runs the repair tool if a process dies before issuing its exit handler and the *xpc* is not running. In fact, *repair* will exit with a message if the *xpc* is happily running. *Repair* will only correct information for processes running on the local node.

For all but the most simple test situations it is best to run the *xpc* and *msg_server* on all nodes rather than running *repair*.

## 7.17 Resetting UNIDAQ

The best way to reset the Unidaq system is to use the Reset tool. This tool will kill all the processes, clear and remove the message queues, shared memory and semaphores used by the system. All machines listed in the $USERUNIDAQ/conf/nodefile file will be reset.

Reset_local is also available to reset only the node on which you are currently running. The machine which is reset need not be listed in the `nodefile` file.

# CONTENTS